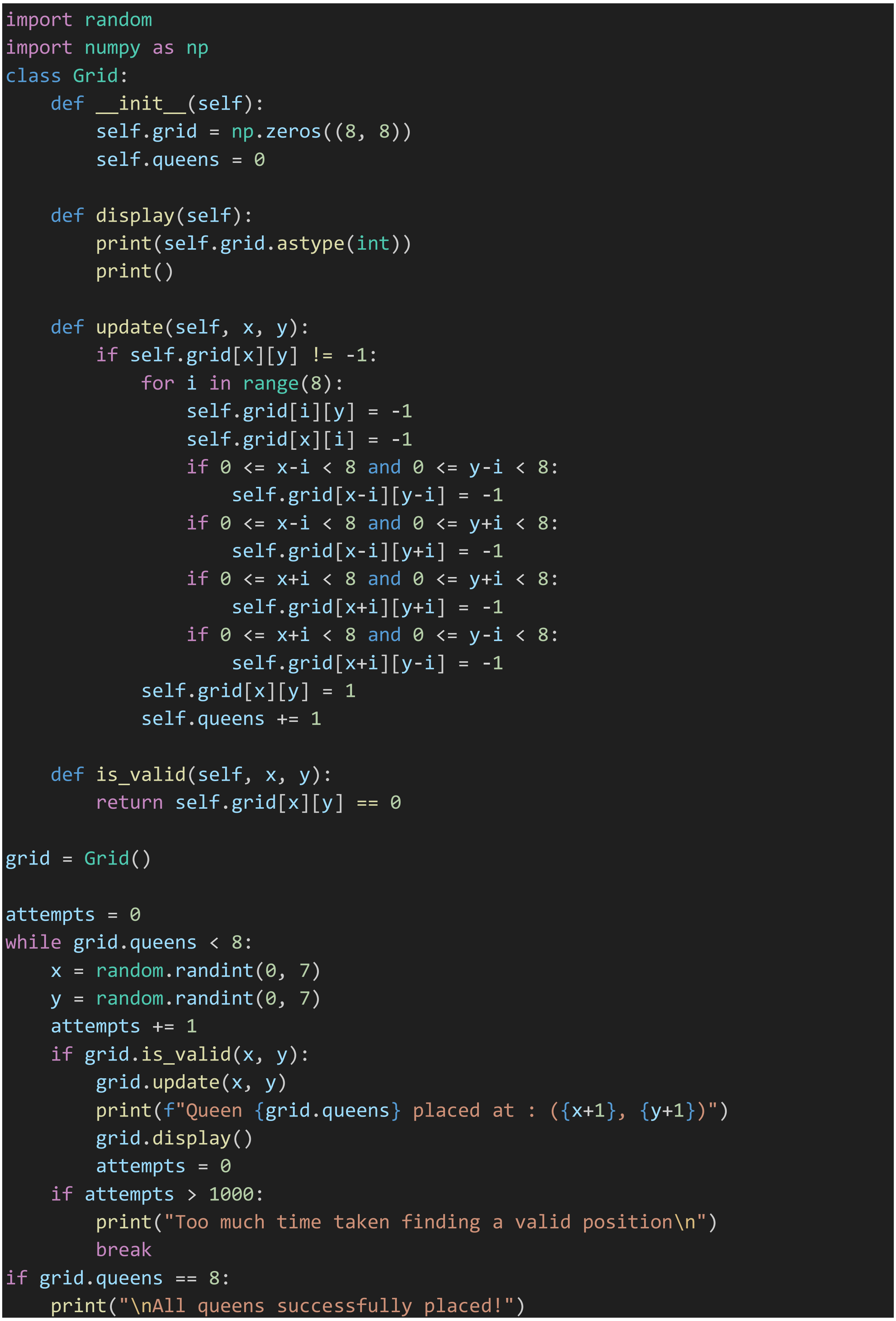
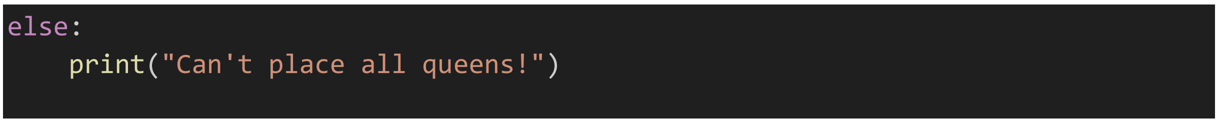
Question 1 and 2:

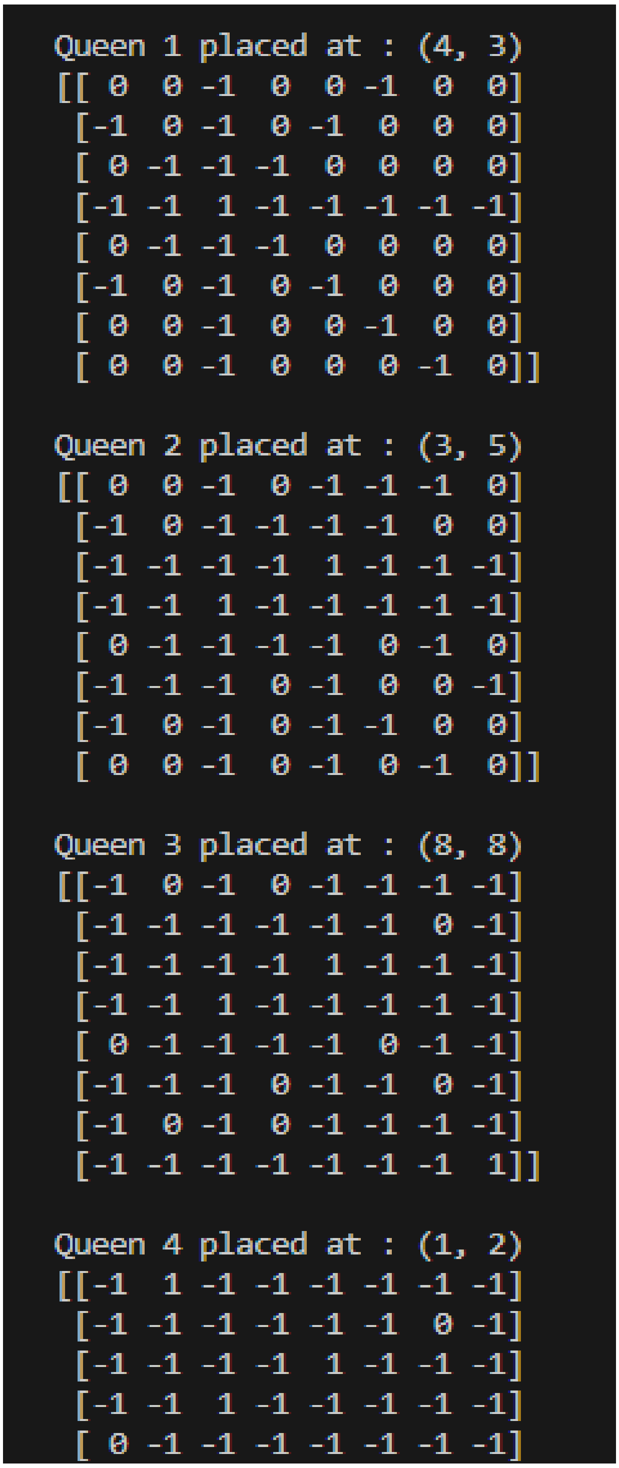


```python
import random
import numpy as np
class Grid:
    def __init__(self):
        self.grid = np.zeros((8, 8))
        self.queens = 0

    def display(self):
        print(self.grid.astype(int))
        print()

    def update(self, x, y):
        if self.grid[x][y] != -1:
            for i in range(8):
                self.grid[i][y] = -1
                self.grid[x][i] = -1
                if 0 <= x-i < 8 and 0 <= y-i < 8:
                    self.grid[x-i][y-i] = -1
                if 0 <= x-i < 8 and 0 <= y+i < 8:
                    self.grid[x-i][y+i] = -1
                if 0 <= x+i < 8 and 0 <= y+i < 8:
                    self.grid[x+i][y+i] = -1
                if 0 <= x+i < 8 and 0 <= y-i < 8:
                    self.grid[x+i][y-i] = -1
            self.grid[x][y] = 1
            self.queens += 1

    def is_valid(self, x, y):
        return self.grid[x][y] == 0

grid = Grid()

attempts = 0
while grid.queens < 8:
    x = random.randint(0, 7)
    y = random.randint(0, 7)
    attempts += 1
    if grid.is_valid(x, y):
        grid.update(x, y)
        print(f"Queen {grid.queens} placed at : ({x+1}, {y+1})")
        grid.display()
        attempts = 0
    if attempts > 1000:
        print("Too much time taken finding a valid position\n")
        break
if grid.queens == 8:
    print("\nAll queens successfully placed!")
```

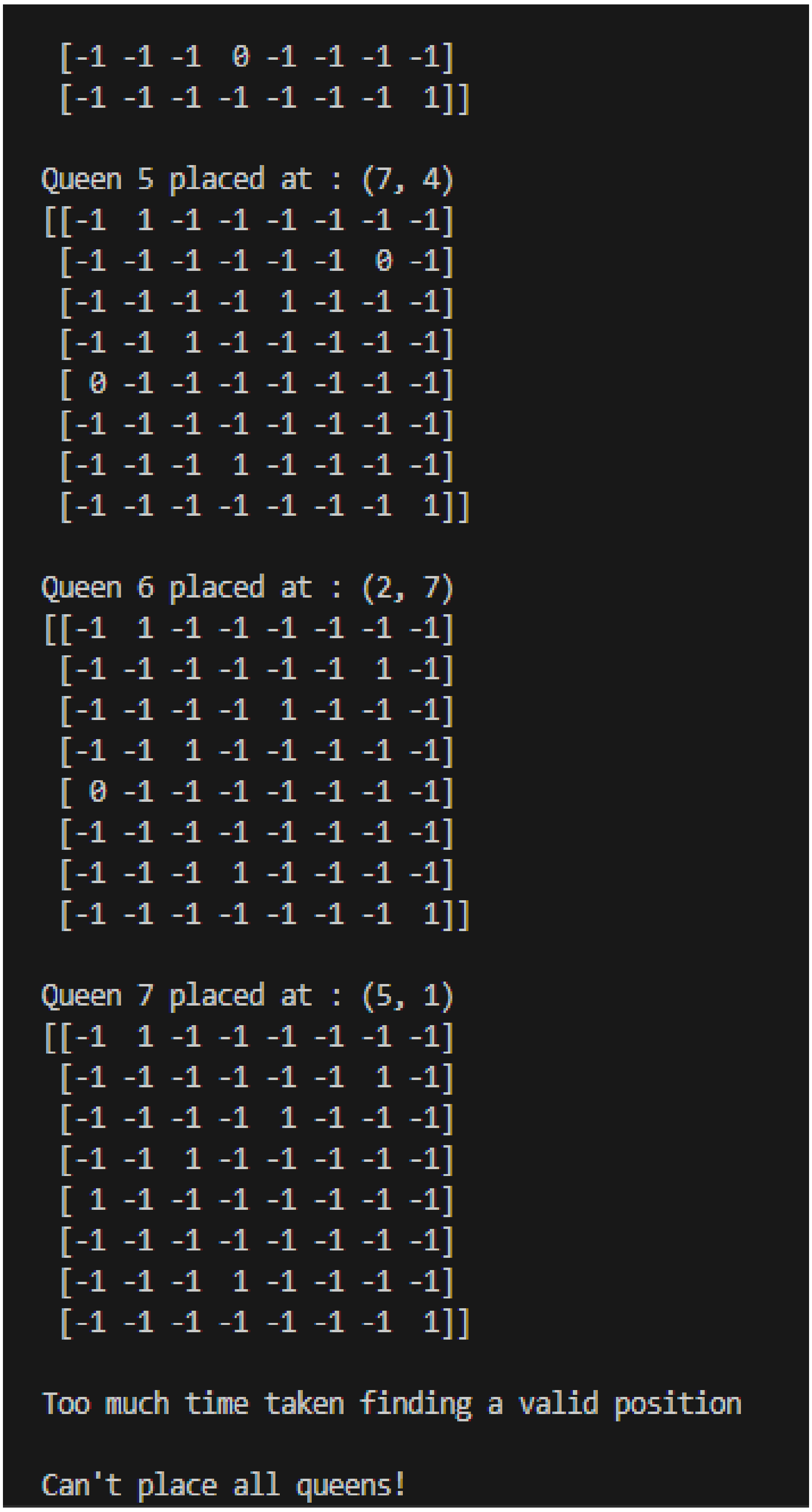


```
else:
    print("Can't place all queens!")
```

Output:

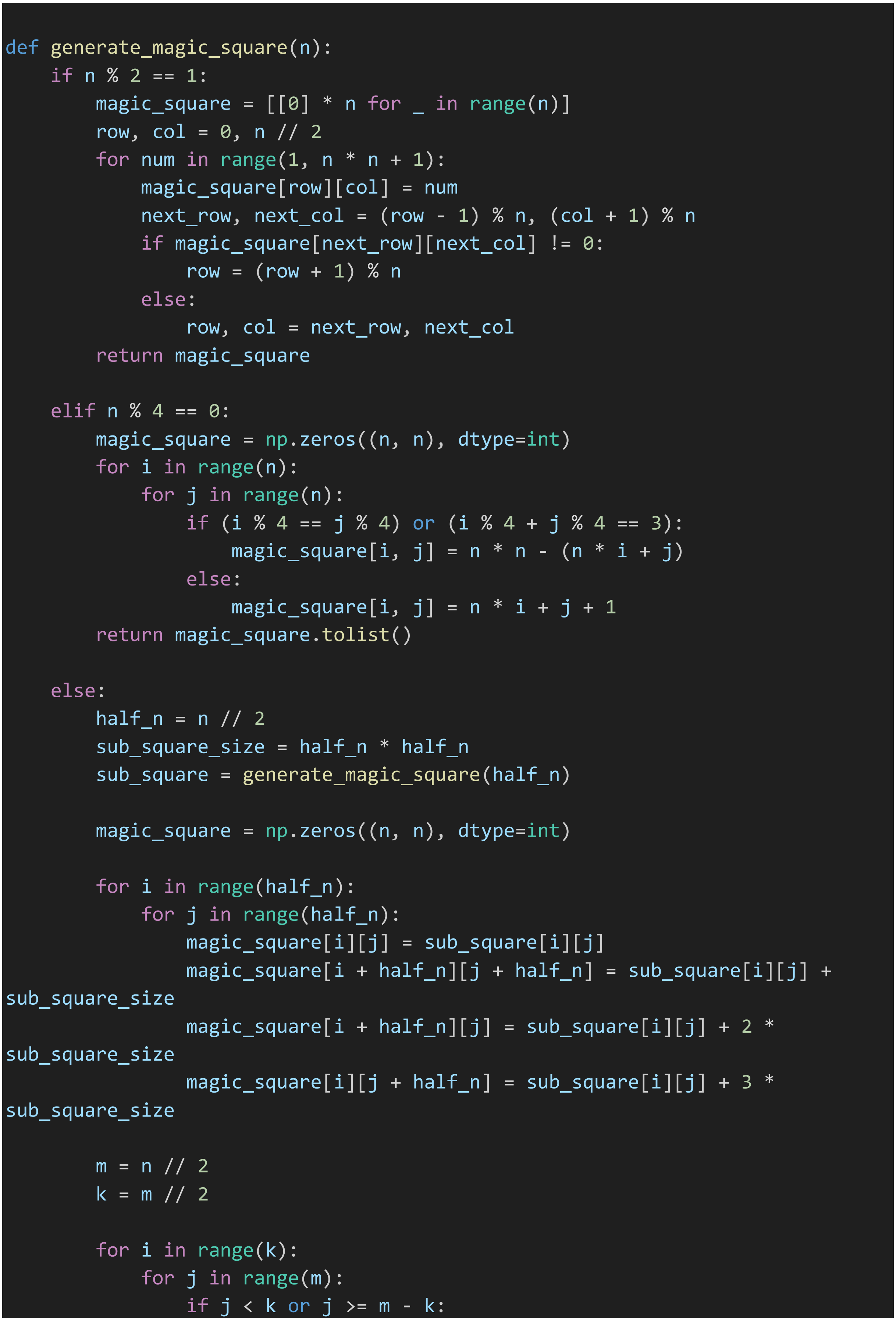


```
Queen 1 placed at : (4, 3)
[[ 0  0 -1  0  0 -1  0  0]
 [-1  0 -1  0 -1  0  0  0]
 [ 0 -1 -1 -1  0  0  0  0]
 [-1 -1  1 -1 -1 -1 -1 -1]
 [ 0 -1 -1 -1  0  0  0  0]
 [-1  0 -1  0 -1  0  0  0]
 [ 0  0 -1  0  0 -1  0  0]
 [ 0  0 -1  0  0  0 -1  0]]

Queen 2 placed at : (3, 5)
[[ 0  0 -1  0 -1 -1 -1  0]
 [-1  0 -1 -1 -1 -1  0  0]
 [-1 -1 -1 -1  1 -1 -1 -1]
 [-1 -1  1 -1 -1 -1 -1 -1]
 [ 0 -1 -1 -1 -1  0 -1  0]
 [-1 -1 -1  0 -1  0  0 -1]
 [-1  0 -1  0 -1 -1  0  0]
 [ 0  0 -1  0 -1  0 -1  0]]

Queen 3 placed at : (8, 8)
[[-1  0 -1  0 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1 -1  0 -1]
 [-1 -1 -1 -1  1 -1 -1 -1]
 [-1 -1  1 -1 -1 -1 -1 -1]
 [ 0 -1 -1 -1 -1  0 -1 -1]
 [-1 -1 -1  0 -1 -1  0 -1]
 [-1  0 -1  0 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1 -1 -1  1]]

Queen 4 placed at : (1, 2)
[[-1  1 -1 -1 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1 -1  0 -1]
 [-1 -1 -1 -1  1 -1 -1 -1]
 [-1 -1  1 -1 -1 -1 -1 -1]
 [ 0 -1 -1 -1 -1 -1 -1 -1]
```



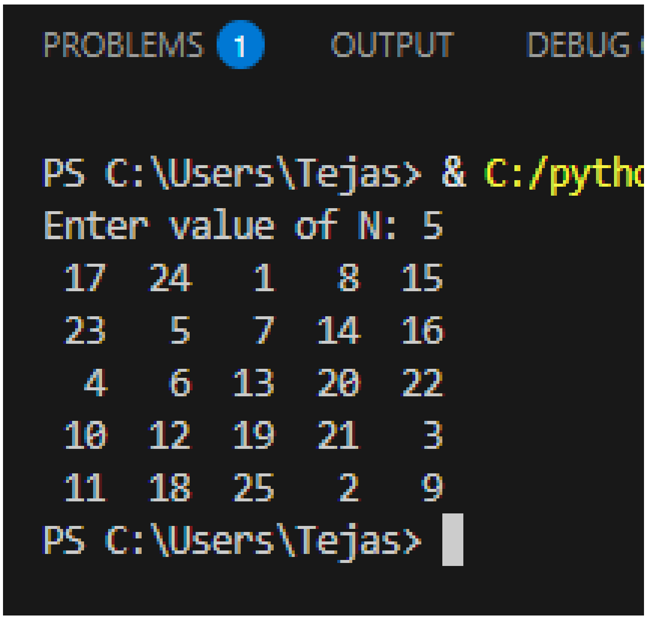
```
 [-1 -1 -1  0 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1 -1 -1  1]]

Queen 5 placed at : (7, 4)
[[-1  1 -1 -1 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1 -1  0 -1]
 [-1 -1 -1 -1  1 -1 -1 -1]
 [-1 -1  1 -1 -1 -1 -1 -1]
 [ 0 -1 -1 -1 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1 -1 -1 -1]
 [-1 -1 -1  1 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1 -1 -1  1]]

Queen 6 placed at : (2, 7)
[[-1  1 -1 -1 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1 -1  1 -1]
 [-1 -1 -1 -1  1 -1 -1 -1]
 [-1 -1  1 -1 -1 -1 -1 -1]
 [ 0 -1 -1 -1 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1 -1 -1 -1]
 [-1 -1 -1  1 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1 -1 -1  1]]

Queen 7 placed at : (5, 1)
[[-1  1 -1 -1 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1 -1  1 -1]
 [-1 -1 -1 -1  1 -1 -1 -1]
 [-1 -1  1 -1 -1 -1 -1 -1]
 [ 1 -1 -1 -1 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1 -1 -1 -1]
 [-1 -1 -1  1 -1 -1 -1 -1]
 [-1 -1 -1 -1 -1 -1 -1  1]]

Too much time taken finding a valid position

Can't place all queens!
```

```python
import numpy as np
```

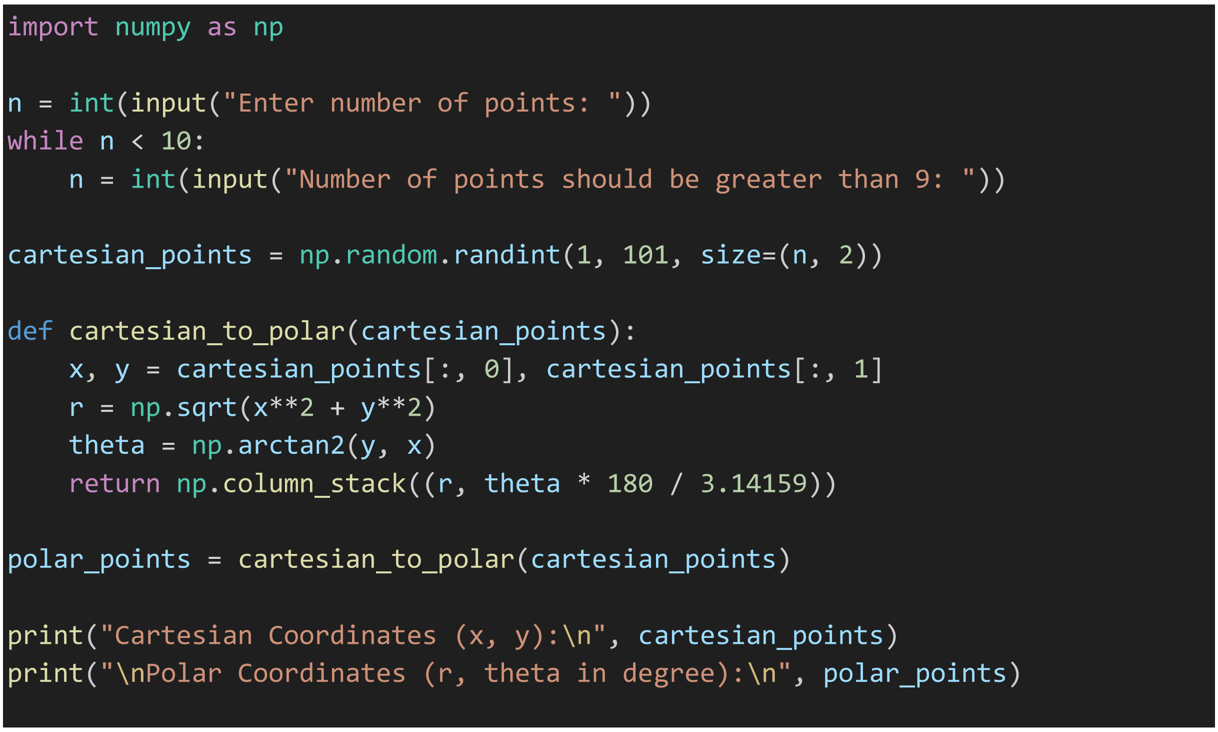


```
def generate_magic_square(n):
    if n % 2 == 1:
        magic_square = [[0] * n for _ in range(n)]
        row, col = 0, n // 2
        for num in range(1, n * n + 1):
            magic_square[row][col] = num
            next_row, next_col = (row - 1) % n, (col + 1) % n
            if magic_square[next_row][next_col] != 0:
                row = (row + 1) % n
            else:
                row, col = next_row, next_col
        return magic_square

    elif n % 4 == 0:
        magic_square = np.zeros((n, n), dtype=int)
        for i in range(n):
            for j in range(n):
                if (i % 4 == j % 4) or (i % 4 + j % 4 == 3):
                    magic_square[i, j] = n * n - (n * i + j)
                else:
                    magic_square[i, j] = n * i + j + 1
        return magic_square.tolist()

    else:
        half_n = n // 2
        sub_square_size = half_n * half_n
        sub_square = generate_magic_square(half_n)

        magic_square = np.zeros((n, n), dtype=int)

        for i in range(half_n):
            for j in range(half_n):
                magic_square[i][j] = sub_square[i][j]
                magic_square[i + half_n][j + half_n] = sub_square[i][j] +
sub_square_size
                magic_square[i + half_n][j] = sub_square[i][j] + 2 *
sub_square_size
                magic_square[i][j + half_n] = sub_square[i][j] + 3 *
sub_square_size

        m = n // 2
        k = m // 2

        for i in range(k):
            for j in range(m):
                if j < k or j >= m - k:
                    magic_square[i, j], magic_square[i + half_n, j] =
magic_square[i + half_n, j], magic_square[i, j]

        return magic_square.tolist()

def print_magic_square(magic_square):
    for row in magic_square:
        print(" ".join(str(num).rjust(3) for num in row))

n = int(input("Enter value of N: "))
if n < 3:
    n = int(input("Value of N can't be less than 3: "))
magic_square = generate_magic_square(n)
print_magic_square(magic_square)
```

Output:

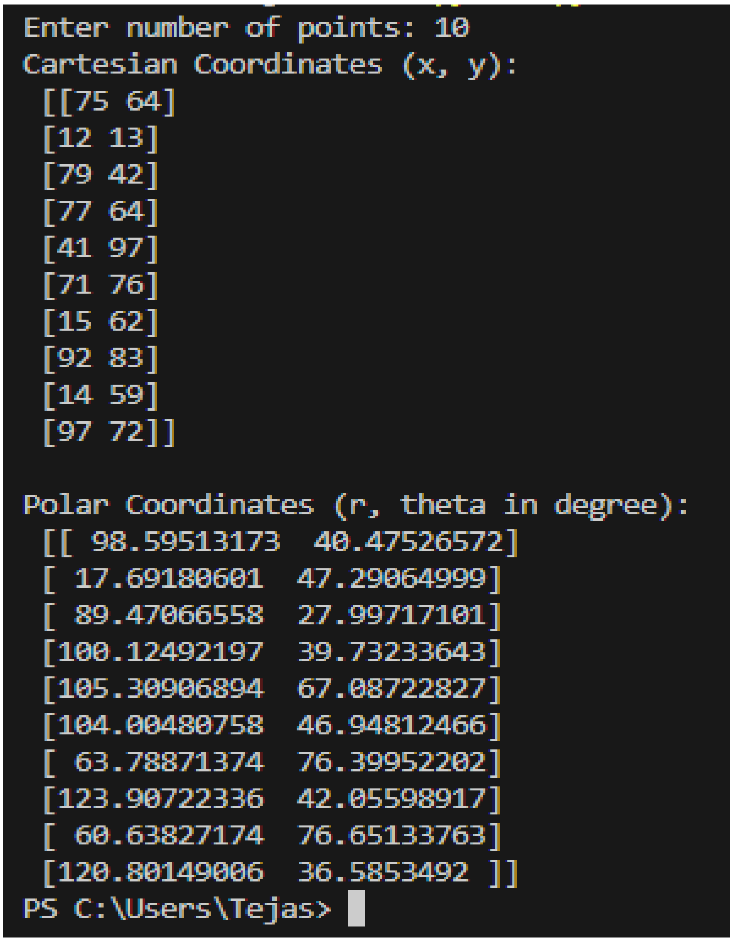


```
PS C:\Users\Tejas> & C:/pytho
Enter value of N: 5
 17  24   1   8  15
 23   5   7  14  16
  4   6  13  20  22
 10  12  19  21   3
 11  18  25   2   9
PS C:\Users\Tejas>
```

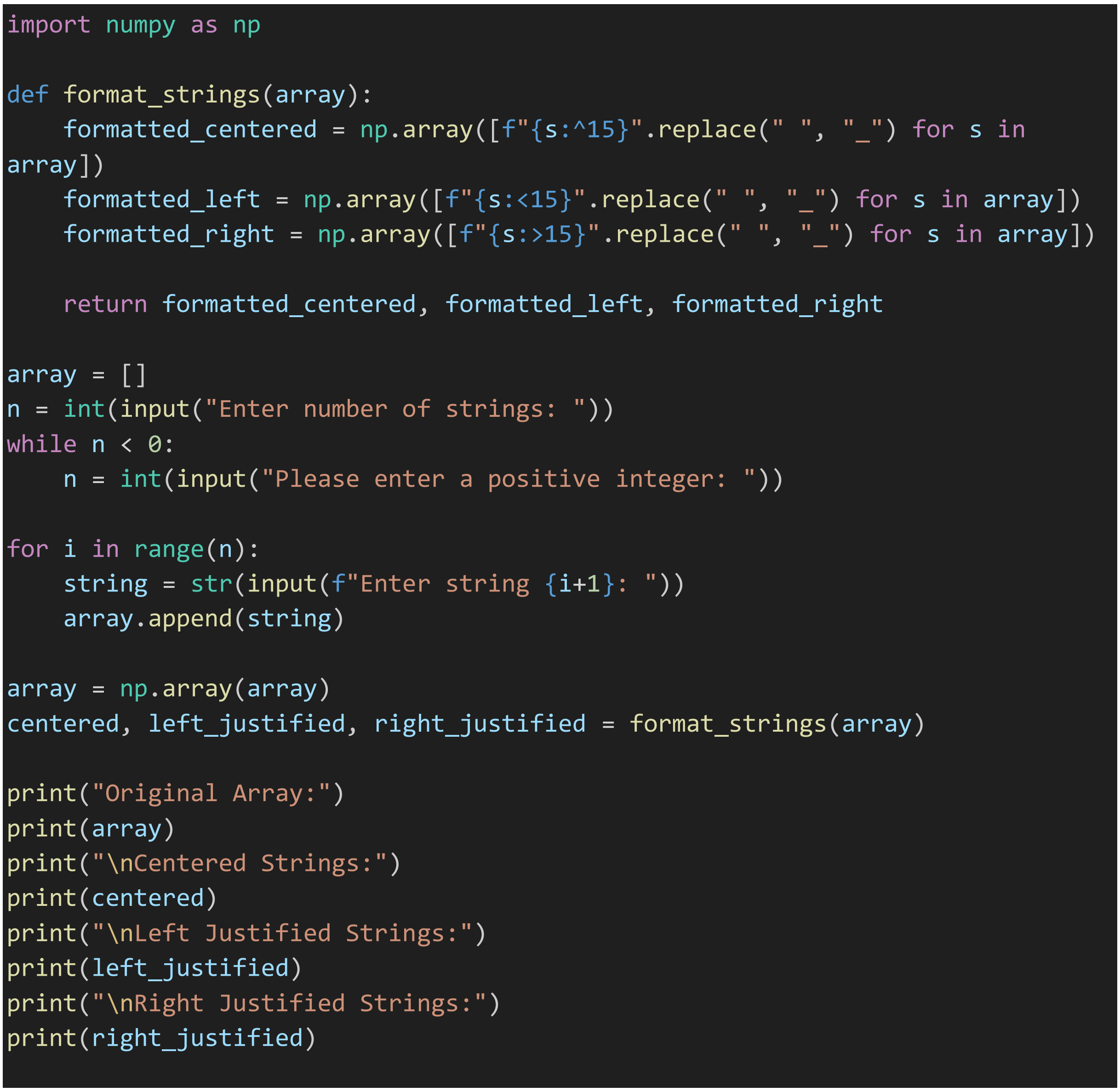


```
import numpy as np

n = int(input("Enter number of points: "))
while n < 10:
    n = int(input("Number of points should be greater than 9: "))

cartesian_points = np.random.randint(1, 101, size=(n, 2))

def cartesian_to_polar(cartesian_points):
    x, y = cartesian_points[:, 0], cartesian_points[:, 1]
    r = np.sqrt(x**2 + y**2)
    theta = np.arctan2(y, x)
    return np.column_stack((r, theta * 180 / 3.14159))

polar_points = cartesian_to_polar(cartesian_points)

print("Cartesian Coordinates (x, y):\n", cartesian_points)
print("\nPolar Coordinates (r, theta in degree):\n", polar_points)
```

Output:

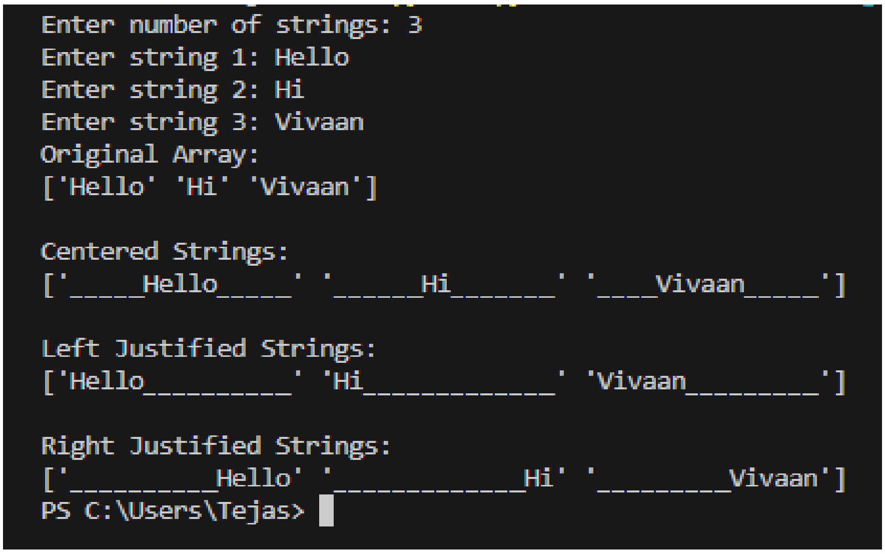


```
Enter number of points: 10
Cartesian Coordinates (x, y):
 [[75 64]
 [12 13]
 [79 42]
 [77 64]
 [41 97]
 [71 76]
 [15 62]
 [92 83]
 [14 59]
 [97 72]]

Polar Coordinates (r, theta in degree):
 [[ 98.59513173  40.47526572]
 [ 17.69180601  47.29064999]
 [ 89.47066558  27.99717101]
 [100.12492197  39.73233643]
 [105.30906894  67.08722827]
 [104.00480758  46.94812466]
 [ 63.78871374  76.39952202]
 [123.90722336  42.05598917]
 [ 60.63827174  76.65133763]
 [120.80149006  36.5853492 ]]
PS C:\Users\Tejas> 
```



```
import numpy as np

def format_strings(array):
    formatted_centered = np.array([f"{s:^15}".replace(" ", "_") for s in
array])
    formatted_left = np.array([f"{s:<15}".replace(" ", "_") for s in array])
    formatted_right = np.array([f"{s:>15}".replace(" ", "_") for s in array])

    return formatted_centered, formatted_left, formatted_right

array = []
n = int(input("Enter number of strings: "))
while n < 0:
    n = int(input("Please enter a positive integer: "))

for i in range(n):
    string = str(input(f"Enter string {i+1}: "))
    array.append(string)

array = np.array(array)
centered, left_justified, right_justified = format_strings(array)

print("Original Array:")
print(array)
print("\nCentered Strings:")
print(centered)
print("\nLeft Justified Strings:")
print(left_justified)
print("\nRight Justified Strings:")
print(right_justified)
```

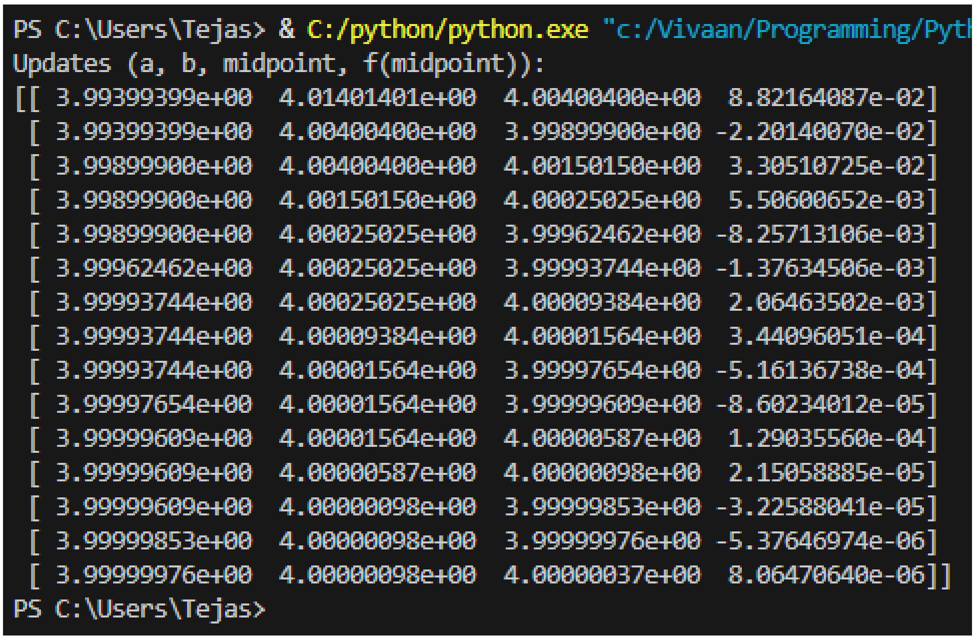
Output:



```
Enter number of strings: 3
Enter string 1: Hello
Enter string 2: Hi
Enter string 3: Vivaan
Original Array:
['Hello' 'Hi' 'Vivaan']

Centered Strings:
['_____Hello_____' '______Hi_______' '____Vivaan_____']

Left Justified Strings:
['Hello__________' 'Hi_____________' 'Vivaan_________']

Right Justified Strings:
['__________Hello' '_____________Hi' '_________Vivaan']
PS C:\Users\Tejas>
```

```python
import numpy as np
import matplotlib.pyplot as plt

def f(x):
    return x**3 - 4*x**2 + 6*x - 24

def bisection_method(f, a, b, tol=1e-6, max_iter=100):
    if f(a) * f(b) > 0:
        raise ValueError("The function must have opposite signs at a and b (f(a)*f(b) < 0).")

    updates = []
    for _ in range(max_iter):
        c = (a + b) / 2
        updates.append([a, b, c, f(c)])
        if abs(f(c)) < tol or (b - a) / 2 < tol:
            break
        if f(c) * f(a) < 0:
            b = c
        else:
            a = c
    return np.array(updates)

np.random.seed(42)
x_probe = np.linspace(-10, 10, 1000)
for i in range(len(x_probe) - 1):
    a, b = x_probe[i], x_probe[i + 1]
    if f(a) * f(b) < 0:
        break

updates = bisection_method(f, a, b)

iterations = np.arange(1, len(updates) + 1)
root_approximations = updates[:, 2]

plt.figure(figsize=(10, 6))
plt.plot(iterations, root_approximations, marker='o', label='Root Approximation')
plt.axhline(0, color='gray', linestyle='--', label='Exact Root Line')
plt.title('Bisection Method Root-Finding Process')
plt.xlabel('Iteration')
plt.ylabel('Root Approximation (x)')
plt.grid()
plt.legend()
plt.show()

print("Updates (a, b, midpoint, f(midpoint)):")
print(updates)
```

Output:



```
PS C:\Users\Tejas> & C:/python/python.exe "c:/Vivaan/Programming/Pyth
Updates (a, b, midpoint, f(midpoint)):
[[ 3.99399399e+00  4.01401401e+00  4.00400400e+00  8.82164087e-02]
 [ 3.99399399e+00  4.00400400e+00  3.99899900e+00 -2.20140070e-02]
 [ 3.99899900e+00  4.00400400e+00  4.00150150e+00  3.30510725e-02]
 [ 3.99899900e+00  4.00150150e+00  4.00025025e+00  5.50600652e-03]
 [ 3.99899900e+00  4.00025025e+00  3.99962462e+00 -8.25713106e-03]
 [ 3.99962462e+00  4.00025025e+00  3.99993744e+00 -1.37634506e-03]
 [ 3.99993744e+00  4.00025025e+00  4.00009384e+00  2.06463502e-03]
 [ 3.99993744e+00  4.00009384e+00  4.00001564e+00  3.44096051e-04]
 [ 3.99993744e+00  4.00001564e+00  3.99997654e+00 -5.16136738e-04]
 [ 3.99997654e+00  4.00001564e+00  3.99999609e+00 -8.60234012e-05]
 [ 3.99999609e+00  4.00001564e+00  4.00000587e+00  1.29035560e-04]
 [ 3.99999609e+00  4.00000587e+00  4.00000098e+00  2.15058885e-05]
 [ 3.99999609e+00  4.00000098e+00  3.99999853e+00 -3.22588041e-05]
 [ 3.99999853e+00  4.00000098e+00  3.99999976e+00 -5.37646974e-06]
 [ 3.99999976e+00  4.00000098e+00  4.00000037e+00  8.06470640e-06]]
PS C:\Users\Tejas>
```

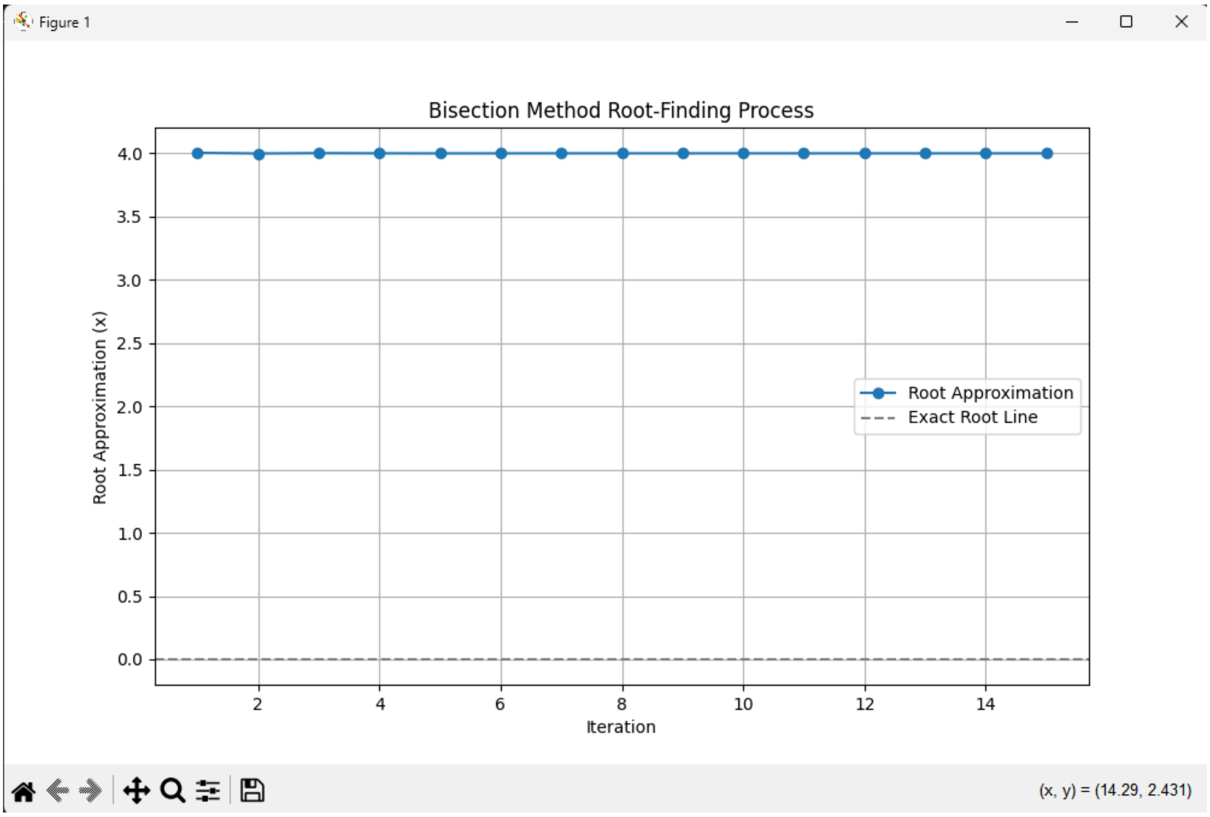


Figure 1
Bisection Method Root-Finding Process
Root Approximation (x)
Iteration
Root Approximation
Exact Root Line
(x, y) = (14.29, 2.431)